Can Graduating Students Design Software Systems?

Ratcliffe, Mark Bartley; Moström, Jan Erik; Eckerdal, Anna; McCartney, Robert; Zander, Carol

Publication date:
2006

Citation for published version (APA):
Ratcliffe, M., Moström, J. E., Eckerdal, A., McCartney, R., & Zander, C. (2006). *Can Graduating Students Design Software Systems?*. http://hdl.handle.net/2160/247

Can Graduating Students Design Software Systems?

Anna Eckerdal
Department of Information Technology
Uppsala University
Uppsala, Sweden
Anna.Eckerdal@it.uu.se

Robert McCartney
Department of Computer Science and Engineering
University of Connecticut
Storrs, CT 06269 USA
robert@cse.uconn.edu

Jan Erik Moström
Department of Computing Science
Umeå University
901 87 Umeå, Sweden
jem@cs.umu.se

Mark Ratcliffe
Department of Computer Science
University of Wales
Aberystwyth, Wales
mbr@aber.ac.uk

Carol Zander
Computing & Software Systems
University of Washington, Bothell
Bothell, WA, USA
zander@u.washington.edu

ABSTRACT

This paper examines software designs produced by students nearing completion of their Computer Science degrees. The results of this multi-national, multi-institutional experiment present some interesting implications for educators.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computers and Information Science Education; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Measurement, Experimentation

Keywords

Design, Software Engineering, Student Performance

1. INTRODUCTION

A fundamental goal of undergraduate computer science programs is that graduates be able to design software systems. This paper will show that this goal is not being met: many students are either unable, or do not properly understand what it means to design a software system. We provide evidence for this finding, offer some explanations why this might be so, and suggest ways that these shortcomings might be addressed.

This research is based on written designs produced by near-graduating seniors, students presumably preparing to work professionally. These designs were collected under stringent interview conditions as part of a larger study, the "scaffolding" experiment: a multi-national, multi-institutional project that looked at the approach students take to software design.

SIGCSE'06, March 1–5, 2006, Houston, Texas, USA.

2. BACKGROUND

Many academics share the opinion that their students' learning of software development is not as effective as it should be. This is evidenced by an international review of first year students' programming skills led by McCracken [5], an often cited example that reports student successful performance in a coding exercise at only 20%. Whilst there has been much research over the years into student coding [7], there is far less available on design. Although we recognize that many other approaches such as Agile Methodologies are now being used to teach software design, recent work by McCracken [6] highlights the poor correspondence between traditional techniques and the cognitive thought processes required when developing software.

Design is recognized to be a difficult topic to comprehend, and as shown by Cross[3], success does seem to require a certain level of cognitive development. Few students reach the highest level where they really appreciate that design principles are context dependent, potentially because they lack sufficient practical experience.

3. THE "SCAFFOLDING" EXPERIMENT

The "scaffolding" experiment[1, 2, 9] was a multi-national, multi-institutional study that looked at several aspects of software design. It included 314 subjects from 21 institutions in the US, UK, Sweden, and New Zealand. As shown in Table 1, the subjects were drawn from three pools ranging from first competency (students who could be expected to program at least one problem from the set proposed by McCracken *et al.*), graduating seniors, and educators.

For the design task, subjects were given a one-page "design brief" (Figure 1) that described the behavior of the desired system: a "super alarm clock" for college students. Subjects were given as much time as they wanted to per-

Table 1: Number of subjects in scaffolding study

Type of subjects	Number
First competency students	136
Graduating seniors	150
Educators	28

Design Brief

Getting People to Sleep

In some circles sleep deprivation has become a status symbol. Statements like "I pulled another all-nighter" and "I've slept only three hours in the last two days" are shared with pride, as listeners nod in admiration. Although celebrating self-deprivation has historical roots and is not likely to go away soon, it's troubling when an educated culture rewards people for hurting themselves, and that includes missing sleep.

As Stanford sleep experts have stated, sleep deprivation is one of the leading health problems in the modern world. People with high levels of sleep debt get sick more often, have more difficulties in personal relationships, and are less productive and creative. The negative effects of sleep debt go on and on. In short, when you have too much sleep debt, you simply can't enjoy life fully.

Your brief is **to design a "super alarm clock" for University students** to help them to manage their own sleep patterns, and also to provide data to support a research project into the extent of the problem in this community. You may assume that, for the prototype, each student will have a Pocket PC (or similar device) which is permanently connected to a network.

Your system will need to:

- Allow a student to set an alarm to wake themselves up.
- Allow a student to set an alarm to remind themselves to go to sleep.
- Record when a student tells the system that they are about to go to sleep.
- Record when a student tells the system that they have woken up, and whether it is due to an alarm or not (within 2 minutes of an alarm going off).
- Make recommendations as to when a student needs to go to sleep. This should include "yellow alerts" when the student will need sleep soon, and "red alerts" when they need to sleep now.
- Store the collected data in a server or database for later analysis by researchers. The server/database system (which will also trigger the yellow/red alerts) will be designed and implemented by another team. You should, however, indicate in your design the behaviour you expect from the back-end system.
- Report students who are becoming dangerously sleep-deprived to someone who cares about them (their mother?). This is indicated by a student being given three "red alerts" in a row.
- Provide reports to a student showing their sleep patterns over time, allowing them to see how often they have ignored alarms, and to identify clusters of dangerous, or beneficial, sleep behaviour.

In doing this you should (1) produce an initial solution that someone (not necessarily you) could work from (2) divide your solution into not less than two and not more than ten parts, giving each a name and adding a short description of what it is and what it does – in short, why it is a part. If important to your design, you may indicate an order to the parts, or add some additional detail as to how the parts fit together.

Figure 1: The design brief given to the subjects

form this task during which they could ask questions of the researcher.

The original study reported significant trends between the groups as they advanced from first-competency to graduating seniors to educators: increasing use of standard graphical representations and decreasing use of text-only descriptions; increased representation of interactions among parts; and increased recognition of ambiguity.

The focus of this study is quite different: how do students design when they are at the end of an undergraduate computing program? To address this question, we undertook a detailed examination of the design artifacts produced by the graduating seniors group.

4. CATEGORIZATION METHODS

The goal of the present study was to examine students' abilities to design software, using their written designs as the primary data. To organize and simplify this data for analysis, we categorized them into groups of similar designs. We chose a data-driven approach for this categorization, with the intention that the categories reflect similarities that we observed in the data. We grouped designs based on their semantics, that is *what* they communicate rather than *how*, together with the extent to which the design met the stated requirement that "someone (not necessarily you) could work from."

Nothing This category has designs with little or no intelligible content. These tend to be very short, typically a single unlabeled diagram.

Restatement These are designs that merely restate requirements from the task description (Figure 1). A typical example is a list of functions that correspond to the bulleted items in that description. These have no design content other than that stated in the description.

Skumtomte[1] These are designs that add a small amount to restating the task. Some subjects added a small amount of information in text, or gave a drawing of a simple GUI with no description of its design, or some unimportant implementation details. There is no overall system view, nor is there any significant work on any of the modules.

First step Designs in this category include some significant work beyond the description: either a partial overview of the system (identifying the parts, but generally not identifying how they are related in the system) or the design of one of the system's components, such as the GUI or the interface to the database.

Partial design A partial design provides an understandable description of each of the parts and an overview of the system that illustrates the relationships between the parts. The descriptions of the parts may be incomplete or superficial and the communication between the parts is not completely described.

Complete design These designs show a well-developed solution, including an understandable overview, part descriptions that include responsibilities, and explicit communication between the parts. A typical example uses multiple formal notations (e.g., UML, Use cases, CRC cards) as well as text.

Figure 2: The six categories used for design artifacts

Based on this approach, after a number of refinements, we identified six categories of designs, shown in Figure 2.

These category descriptions included a general definition and to help to clarify ambiguous designs, referred to a typical *prototype*, one of the designs in the dataset. This provided a mechanism for dealing with "fuzzy" boundary cases that might otherwise have been difficult to categorize.

The process of developing the categories and then assigning the designs was data-driven and integrated—both the category descriptions and the previous design assignments evolved as the category assignment (*tagging*) progressed.

[1]The Swedish word *Skumtomte* refers to a pink-and-white marshmallow Santa Claus, a traditional Christmas confection. It looks like there is something there, but it is only shaped and colored marshmallow fluff.

4.1 Comparison with other studies

After developing and refining these categories, we considered them relative to relevant computer science education research.

DuBoulay [4] comments on novice programmers' inability to grasp the whole program and its constituent parts: "This ability to see a program as a whole, understand its main parts and their relation is a skill which grows only gradually.". This is in line with our conclusions that overview of the parts and relations between parts are important characteristics found only in the more advanced designs.

Soloway *et al.* [8] discuss how to teach design. Based on a study with expert software designers the authors advocate five phases in developing a design. The summarized phases are

- *Phase 1: Understand problem specification.* The goal here is simply to understand what the problem is asking for.
- *Phase 2: Decompose problem into programmable goals and objects.* During this phase, the objective is to "lay the components of the solution on the table", that is, decompose the problem and identify the solution components.
- *Phase 3: Select and compose plans to solve problems.* During this phase the pieces of the solution are woven together, that is, the components are composed to form a working whole.
- *Phase 4: Implement plans in language constructs.*
- *Phase 5: Reflect–Evaluate final artifact and overall design process.* When all is said and done, a good strategy is to look back over what has been done and learn from both the successes and failures.

It is interesting to compare Soloway's phases with the data driven classification in this study. Soloway's first phase can be directly mapped to our *Restatement* and *Skumtomte* groups where the designs indicate that the students were trying to understand the problem. It is worth noting that the majority of the designs did not progress past the first of Soloway's phases.

The second phase, to "lay the components of the solution on the table," we interpret as getting an overview of the solution. The overview sometimes occurs in the *First step* solutions, but is always found in our two top categories. These designs show an understanding of the problem and display the components of the solution.

In the third phase, "the pieces of the solution are woven together." This assumes that both the responsibilities and relations (connections and communications) among the parts are determined, things which are part of the top two categories in our study.

The fourth and fifth phases, implementation and evaluation, are not within the scope of the task given in our study. However, there were examples of code fragments or other implementation decisions (fourth phase), found in all categories except *Nothing*, with particularly strong evidence in the *Skumtomte* and *First step* design categories. There were also designs in the top groups that showed reflection and evaluation of their solutions, and so illustrated Soloway's fifth phase.

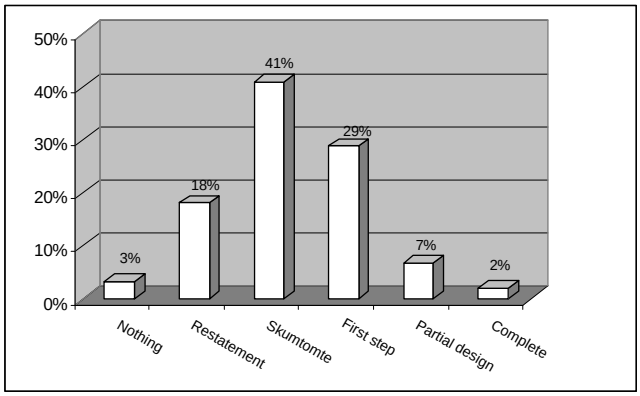

Figure 3: Frequencies of observations in each of the design categories (based on 149 observations)

Soloway *et al.* describe the phases as "activities" and that "there may be (will be!) some jumping around – back and forth" based on the results from the study with the expert designers. This agrees with our design data. We do not see a monotonic progression through the phases in the written designs and we see traces of many of these phases throughout the documents.

5. RESULTS AND OBSERVATIONS

The overall performance is illustrated by the frequency plot shown in Figure 3. On the whole, this is quite negative:

- 21% of the designs were simply restatements of the specification or less—no value added at all.
- 41% of the designs were *Skumtomte*: those that added an insignificant amount beyond the specification, and, in particular, did not produce any "design content."
- 29% of the designs were in the *First step* category, showing some progress toward a design—a partial overview, or significant progress toward the design of one part of the system.
- 9% produced "reasonable" designs *(Partial design* or *Complete)*: those including an understandable system architecture, with parts and their interactions explicitly stated. Of these, less than a third produced *Complete* designs, with explicit part responsibilities and inter-part communications.

All in all, a poor performance from students who are near graduation: over 20% produced nothing, and over 60% communicated no significant progress toward a design.

Having categorized the designs, however, we can examine some questions more closely. Specifically, are there other factors that correlate with design performance such as age, gender, academic record, and time spent on the design task? We consider these below.

5.1 Correlation with other factors

As part of the overall study, we collected academic and demographic background data on the students and made other observations during the design task. To further understand which of these might be related to design performance, we contrasted these data with the design categories.

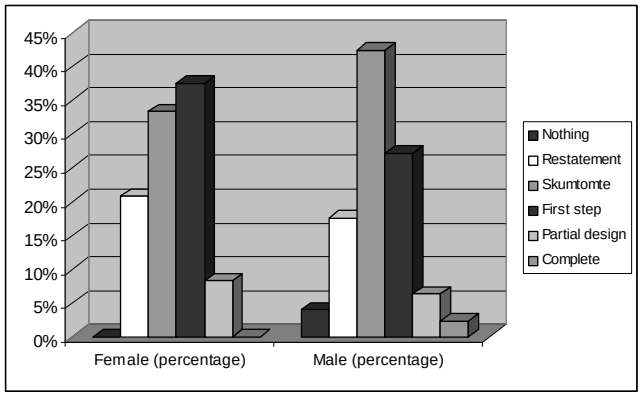

Figure 4: Frequencies per category by gender

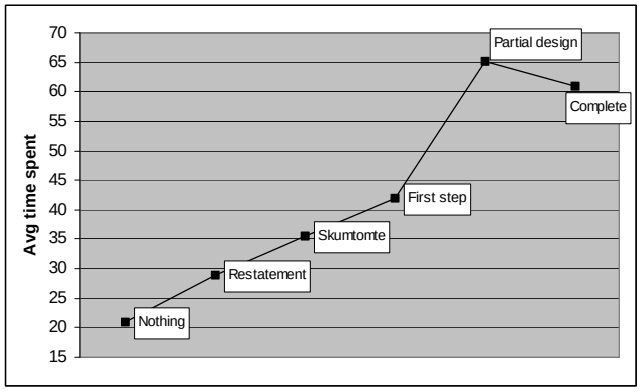

Figure 5: Average time spent on the problem

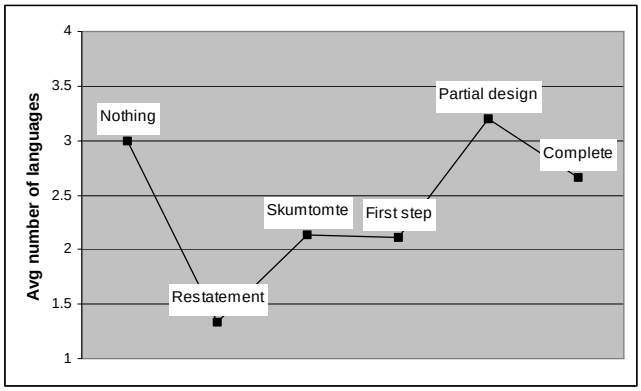

Figure 6: Languages that a students knows "well"

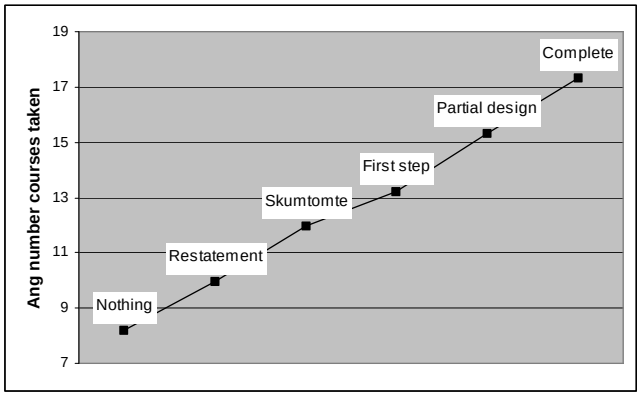

Figure 7: Average number of CS courses completed

We summarize the results as follows.

Age. Most participants were in the range 20-27, but other age groups did not perform better or worse than the majority.

Gender. As seen in Figure 4, when the designs were mapped as a percentage in relation to gender, we observe a number of differences. The males created all the designs at the extremes, in the *Nothing* and *Complete* design categories. In terms of producing added design content, females did better with 46% of their designs in the top three categories, while the males had 36% of theirs in the top three categories. This difference is nearly all attributable to the females producing relatively more *First step* designs, and the males producing relatively more *Skumtomte* designs.

Time. If we plot the average time spent on the exercise, we see that the students who took more time generally produced better designs (Figure 5).

Number of languages. The number of languages a student has used does not seem to affect the result. It should be noted that this measurement is very crude since it may include languages that have been used for a very short amount of time, for example, two hours of C during a course in electronics.

Familiarity. The students were also asked to indicate their "familiarity" with the different languages they had used on a scale ranging from 1 (never used) to 5 (have used a lot). Figure 6 shows that (with the exception of the *Nothing* group, which had only five designs), as the number of "familiar" languages (rated 4 or 5 by the student) increases, students tend to produce better designs.

Number of courses. Figure 7 shows a positive correlation between the number of CS courses taken and the category of the design. More courses could mean a stronger interest in the subject; it at least indicates more experience with the material.

Academic grade. Academic performance, as measured by grade point average for computer science courses, seems to have little or no relationship to the design produced as seen in Figure 8. In fact, the students that produced the best designs had grades that could be classified as "average." Many of the top performing students created designs that were classified into the groups *Restatement*, *Skumtomte*, and *First step*.

6. IMPLICATIONS FOR EDUCATORS

When the designs were studied in detail and categorized, we found that certain features characterize the more advanced designs: an overview, details on part responsibilities, and communication between the parts.

An overview is of great importance for a good understanding of the design produced. The design brief explicitly stated

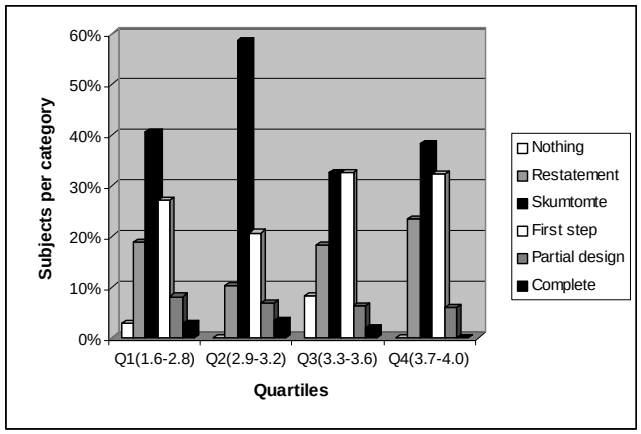

Figure 8: Academic performance

that "someone (not necessarily you)" should be able to work from the design produced. This idea is closely connected to Soloway's description of the first phase of a design, to understand the problem specification [8]. To decompose the problem and weave the parts together are more advanced procedures as pointed out by DuBoulay [4].

Quite significantly, a number of designs in the *Skumtomte* and *First step* groups are long relative to the substance that they communicate. 55% of the designs in the *Skumtomte* group and 67% of the designs in the *First step* group (the same number as in the *Complete* design group) are three pages or longer. Despite considerable effort put into the design, these students end up with little. If we could understand what it was that these students were trying to achieve, we might better understand what they consider significant.

The *Skumtomte* group neglected both the overview and communication aspects, but may have been distracted on unimportant details. Even if the design process is a "jumping back and forth" as pointed out by Soloway *et al.*, a novice designer still needs to understand what aspects of the design process belong to an early stage of the process and what can be delayed.

7. CONCLUSIONS

The results of this study show that the majority of graduating students cannot design a software system. Taking more courses in computer science seems to improve design technique, as does having significant experience with more programming languages. Surprisingly perhaps, academic performance in general does not indicate how well the student can design software. Most of the students in this study did not seem to understand what sort of information a software system design should include, and how that information might be communicated.

These results must be taken in context though. In school, students know what is expected by the course they are in, and their instructions are usually made to be as clear as possible. For example, they might be told to produce a software design document and be given an outline of what the document is to contain. If a UML class diagram or a sequence diagram is desired, students are told to produce these. Can students be expected to produce a solid software design without explicit instructions when they have not done so previously? As instructors, we expect students to be able to take this leap on their own. Perhaps we are wrong, and should offer students more experience in dealing with underspecified tasks.

8. ACKNOWLEDGMENTS

The authors would like to thank Sally Fincher, Marian Petre, Josh Tenenberg, the Scaffolding workshop participants, and the National Science Foundation (through grants DUE-0243242 and DUE-0122560) for their support and encouragement.

9. REFERENCES

[1] K. Blaha, A. E. Monge, D. Sanders, B. Simon, and T. VanDeGrift. Do students recognize ambiguity in software design? a multi-national, multi-institutional report. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 615–616, 2005.

[2] T. Chen, S. Cooper, R. McCartney, and L. Schwartzman. The (relative) importance of software design criteria. In *Proceedings of the 10th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2005)*, pages 34–38, Monte da Caparica, Portugal, June 2005.

[3] P. Cross. What do we know about students learning and how do we know it. *AAHE National Conference on Higher Education*, 1998.

[4] B. DuBoulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1):57–73, 1986.

[5] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin*, 33(4):125–180, 2001.

[6] W. M. McCracken. Research on learning to design software. In S. Fincher and M. Petre, editors, *Computer Science Education Research*. Taylor and Francis Group, London, 2004.

[7] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137 – 172, 2003.

[8] E. Soloway, J. Spohrer, and D. Littman. E unum pluribus: Generating alternative designs. In R. E. Mayer, editor, *Teaching and Learning Computer Programming*, pages 137–152. Lawrence Erlbaum Associates, Publishers, 1988.

[9] J. Tenenberg, S. Fincher, K. Blaha, D. Bouvier, T. Chen, D. Chinn, S. Cooper, A. Eckerdal, H. Johnson, R. McCartney, A. Monge, J. Moström, M. Petre, K. Powers, M. Ratcliffe, A. Robins, D. Sanders, L. Shwartzman, B. Simon, C. Stoker, A. Tew, and T. VanDeGrift. Students designing software: a multi-national, multi-institutional study. *Informatics in Education*, 4(1):143–162, 2005.